(2)

AD-A221 677

The RHET Plan Recognition System
Version 1.0

Bradford W. Miller

Technical Report 298
January 1990

UNIVERSITY OF
ROCHESTER
COMPUTER SCIENCE

The RHET Plan Recognition System Version 1.0

Bradford W. Miller

The University of Rochester
Computer Science Department
Rochester, New York 14627

Technical Report 298

January 1990

Abstract

RPRS is a hierarchical plan recognition system built within the RHET knowledge representation system. It provides a powerful system for plan recognition based on the algorithms of Kautz[Kautz, 1987], with the general reasoning capabilities of RHET. RPRS takes special advantage of Rhet's type relations, constraints, equality and contextual reasoning abilities.

RPRS is also intended as a demonstration of the Rhet programming and knowledge representation system's hybrid reasoning capabilities. Utilizing the lisp interface to Rhet, RPRS allows the user to use the Rhet structured type system to build plan types, and given some observation or set of observations have Rhet derive the set of plans that are consistent with these observations. Since RPRS includes the TEMPOS specialized reasoner for Rhet, steps and observations can have reference to time-intervals, and/or be temporally constrained with respect to one another.

This work was supported in part by ONR research contract no. N00014-80-C-0197, in part by U.S. Army Engineering Topographic Laboratories research contract no. DACA76-85-C-0001, and in part by the Air Force Systems Command, Rome Air Development Center, Griffiss Air Force Base, New York 13441-5700, and the Air Force Office of Scientific Research, Bolling AFB, DC 20332 under Contract No. F30602-85-C-0008. (This contract supports the Northeast Artificial Intelligence Consortium (NAIC).)

Contents

List of Figures

Chapter 1

Introduction

The RHET Plan Recognition System (RPRS) is a simple yet powerful plan recognition system built upon the Rhet [Allen and Miller, 1989][Miller, 1989] and TEMPOS [Koomen, 1989][Allen and Miller, 1989][Koomen, 1988] systems. The user interface to this system is intended to be very similar to working directly with Rhet, and serves as an example of writing a layered system (in lisp) on Rhet.

Please note that this document is NOT meant to be a manual for RHET, TEMPOS, or TIMELOGIC. An overview of these systems is given below, however understanding the details of the examples will require the reference documentation as cited above.

The basic idea is as follows: the user constructs structured types that describe *plans*, using Rhet's structured type definition functions and those RPRS provides. These plans may have other plans as *steps*, or actions. (In fact, there are two types of plans, those that can be recognized, and those that are only useful as constituent steps of other plans). The user then presents a list of one or more observed actions to RPRS, which will then return a set of contexts and plan instances in these contexts, each of which satisfy all of the observed actions. The algorithm for doing this is loosely based on [Kautz, 1987], however his system worked hierarchically and could do graph matching. while RPRS treats every recognizable plan type (regardless of if it has subtypes) as a separate recognizable plan and works with contexts instead of trees. It is therefore an offline rather than online algorithm[1]. The advantage of building the system on Rhet (and TEMPOS) has been the ability to use the much more powerful notions of structured types, equality, inequality, and time reasoning that these systems supply. For example, we can construct plans whose agents must be blood relatives, have eaten dinner within some period of time of each-other, and not have any offspring in common; we can have complex relationships between the steps of a plan, *e.g.* that to get a discount flight one must book one's tickets two weeks in advance of the departure time, or that for two events, both of which must be done during daylight, the first

[1] RPRS could be enhanced to be online, but the cost may not save enough over the offline algorithm to make such development worthwhile; in particular considerably more state would have to be maintained between proofs.

one must precede the second. Constraining the first to occur just before nightfall would force the second to wait until (at least) the next morning.

RPRS is supplied with a demonstration script that shows a variety of examples of recognizing plans given certain constraints, ambiguous situations, *etc.*.

1.1 An Abbreviated Introduction to Rhet

Rhet is a hybrid Knowledge Representation system that offers a set of tools for building automated reasoning systems. In overview, one can think of it as COMMON LISP, PROLOG, E-Unification, and a frame language such as KL1 having a head-on collision at 420 miles per second (mps). There are several kinds of knowledge one can assert in Rhet.

Factual One can assert, for instance, that `[P A]` or that `[NOT [P B]]`. This can also be done relative to a context, *e.g.* `[MB [P A]]` can be read as "it is mutually believed among all agents that `[P A]`".

Deductive / Backward Chaining One can assert in Horn Clause form, things that can be inferred when a proof is attempted, *e.g.* `[[Q ?x] < [P ?x]]`. Given the above assertions, a proof of `[Q B]` would succeed. Contexts can also be used in these expressions.

Constrained Once can postpone proofs or generally constrain variables such that some predicate is true. This avoids the possible performance inefficiency of backtracking by only proving the predicate is true of some particular binding, rather than the usual approach of backtracking through all possible bindings of the variable until one allows the predicate to succeed. Since a constrained variable can also be a valid proof result, one can represent what would otherwise be an infinite set.

Inductive / Forward Chaining One can assert in something similar to Horn Clause form things that can be asserted when factual information is added, *e.g.* `[[IS-BIRD ?x] < [IS-PENGUIN ?x] :forward]]` would assert IS-BIRD is true for any object that IS-PENGUIN is asserted of. Again, these rules may be contextual.

Equality One can assert (Add-Eq [MOTHER-OF SAM] [DEBORAH]), which would allow, *e.g.* `[BAKES-COOKIES DEBORAH]` to be provable if `[BAKES-COOKIES [MOTHER-OF SAM]]` had been asserted. More generally, inequality can be added, *e.g.* (Add-Ineq [MOTHER-OF JOE] [MOTHER-OF SAM]) which would not only disallow future assertion that they were equal, but that JOE and SAM are equal since then the two MOTHER-OF terms above would be equal. Equality and inequality are context sensitive.

Type Structured Types (frames) can be defined. These are extremely flexible and powerful. In general a type may have one or more roles defined on it. For instance, one might define a T-PERSON type with roles R-MOTHER, R-FATHER, *etc.*. These roles are

accessed using accessor functions like [F-MOTHER ?p*T-PERSON]. It is possible to define constraints between the roles of a type, *e.g.* `[NOTEQ? [F-MOTHER ?self] [F-FATHER ?self]]`, as well as setting up type-specific arbitrary relations with other structured types, e.g. that for objects of type T-TRAVEL-BY-AIRPLANE a step will be an instance of type T-BUY-AIRLINE-TIX.

Further, functions can have their result type depend on their argument types. See the examples below.

Procedural The result of a LISP function can be used as a predicate result or bound to a variable.

Let's look at some simple examples (for the most part, taken from [Allen, 1990]: First constrained variables...Assert-Axioms simply asserts it's arguments to be true.

(1.1) `(ASSERT-AXIOMS [[P A] <])`

Unify attempts to do standard unification between it's two arguments. Note the first argument in 1.2 is a constrained variable, here ?x is constrained such that `[P ?x]` must be true. 1.2 should fail since we can't prove `[P B]`.

(1.2) `(UNIFY [ANY ?X [P ?X]] [B])`

while 1.3 should succeed since `[P A]` was asserted.

(1.3) `(UNIFY [ANY ?X [P ?X]] [A])`

Because Rhet will both try to Prove and Disprove a goal (or any subgoal using Complete reasoning) it can do simple consistency tests. 1.4 adds rules and facts to the default SBMB[2] context, which is inherited by the SB context.

```
(1.4)  (ASSERT-AXIOMS [[Q] < [R]]
                      [R <]
                      [[NOT R] < [S]])
```

Simple proofs do not try to do a disproof. 1.5 and 1.6 will both fail, that is, in the SB context, we can't prove `[NOT Q]` or `[NOT R]` because neither has been asserted, nor do they appear on the LHS of a provable rule; so 1.7 will succeed. Had either been provable it would have failed since it checks on each proof level for inconsistency; that is proving [Q] will cause a subproof of `[NOT Q]` which will fail, and then invoke the first axiom above, causing a subproof of `[R]`. Similarly this will invoke a subproof of `[NOT R]`, which matches an unprovable axiom, and so fails, and then the term `[R]`, which has been asserted, is found and causes the open proofs to succeed.

[2]Or "System believes, it is Mutually believed among all agents". MB is only accepted as a trailing token in belief operators, and only works among all agents, otherwise SB stands for System believes, and is special, any other token, *e.g.* AB, HB, stand for that agent's beliefs, here A or H. SBHBSB would read the system's beliefs about H's beliefs about the system's beliefs. For the purposes of these examples, one only need to know that SB inherits everything in SBMB, but SBMB inherits nothing from SB.

(1.5) (PROVE [SB [NOT Q]] :SIMPLE)

(1.6) (PROVE [SB [NOT R]] :SIMPLE)

(1.7) (PROVE [SB [Q]] :COMPLETE)

Now introduce an inconsistency. This will allow [NOT R] to be provable.

(1.8) (ASSERT-AXIOMS [S])

1.9 will succeed since the predicate Q is consistent, but 1.10 will fail because the proof of Q involves an inconsistency. That is, only 1.10 will attempt to prove [NOT R], succeed, and so fail.

(1.9) (PROVE [SB [Q]])

(1.10) (PROVE [SB [Q]] :COMPLETE)

OK, remove the inconsistency.

(1.11) (RETRACT [S])

Rhet can handle retractions on a contextual basis. 1.12 will retract [R] from the SB context, and assert [NOT R] there in its place.

(1.12) (ASSERT-AXIOMS [[SB [NOT R]] <])

Now 1.13 will fail while 1.14 will succeed. From the above example, you might expect them both to succeed, because 1.13 doesn't involve a proof of [NOT R], since it isn't a complete proof. But what has happened is that in context SB, [R] has actually been retracted! So it isn't the case we can prove both the [R] that is inherited from SBMB, and the [NOT R] local to SB, but only the [NOT R] in SB.

(1.13) (PROVE [SB [Q]])

(1.14) (PROVE [Q])

This example will define a function SUM whose arguments are always of type integer, but the object [SUM ?x ?y] is of type EVEN if both ?x and ?y are of type EVEN or of type ODD, and of type ODD if ?x and ?y are of different types. Declare-FN-Type is how we declare some function takes arguments of some type and produces a result type. The first typelist is special; it declares that the arguments will never be supertypes of these types, and in fact, that Rhet is free to change the type of a term that this function is called on appropriately.

```
(1.15) (DECLARE-FN-TYPE 'SUM
                        ;; useful maxtypes
                        '(T-INTEGERS T-INTEGERS T-INTEGERS)
                        '(EVEN EVEN EVEN)
                        '(ODD ODD EVEN)
                        '(EVEN ODD ODD)
                        '(ODD EVEN ODD))
```

Now 1.16 will cause, for instance, ?x to be constrained to give result 1.17. Note that ?x and ?y were changed to be of type T-Integers since that was the maxtype defined for SUM.

```
(1.16) (UNIFY [P [SUM ?X ?Y]] [P ?Z*ODD])
```

```
(1.17) [ANY ?X*T-INTEGERS
            [TYPE? [SUM ?X*T-INTEGERS
                        [ANY ?Y*T-INTEGERS
                             [TYPE? [SUM ?X ?Y]
                                    *ODD ]]]
                   *ODD ]]
```

Last, a small structured type example:

```
(1.18) (DEFINE-SUBTYPE 'T-PARENT 'T-PERSON :ROLES '((R-CHILD T-PERSON)))
```

```
(1.19) (DEFINE-SUBTYPE 'T-GRAND-PARENT 'T-PARENT
                       :ROLES '((R-GRANDCHILD T-PERSON)
                                (R-CHILD T-PARENT))
                       :CONSTRAINTS '([EQ? [F-GRANDCHILD ?SELF]
                                           [F-CHILD [F-CHILD ?SELF]]]))))
```

1.18 defines a new structured type T-PARENT which inherits from type T-PERSON and has role R-CHILD also of type T-PERSON. 1.19 then defines a grandparent as a parent with a grandchild role. It specializes the type of the inherited child role to be a parent, and constrains the grandchild role to be the child of it's child role. Now we will create a couple of instances of grandparents; the idea here is we want to see that Rhet maintains the correct relationships between grandparents, parents, and children.

```
(1.20) (DEFINE-INSTANCE [G1] 'T-GRAND-PARENT)
```

```
(1.21) (UTYPE 'T-PERSON [G3])
```

```
(1.22) (DEFINE-INSTANCE [G2] 'T-GRAND-PARENT
                        :R-CHILD [G1]
                        :R-GRANDCHILD [G3])
```

Now when we look at the equivalence class of [G3] we will find it is the child of [G1], the child of the child of [G2], and the grandchild of [G2].

(1.23) (EQUIVCLASS [G3])

1.2 An Abbreviated Introduction to TEMPOS

The ability to reason about time intervals, as proposed by Allen [Allen, 1983], is an important adjunct to the planning process. Temporal constraints can be manipulated independently of other planning constraints, which allows for the planner to deal with such subtleties as reasoning about past or future events, dealing with partially constrained ordering between plan steps (as opposed to the instantaneous sequential ordering of a STRIPS like system, as in [Nilsson, 1980]). This is what motivated our desire to build a planning system that can take advantage of the temporal world model as elaborated in [Allen and Koomen, 1983].

TEMPOS extends the reasoning capabilities of Rhet by including hooks and builtins that cleanly allow Rhet to interface to the Timelogic system. Whenever the user constructs an object of type T-TIME, the TEMPOS system intercepts the construction and causes Timelogic to be aware of it. Similarly, should the user or Rhet internally assert two objects of type T-TIME to be equal, TEMPOS will cause Timelogic to appropriately constrain the intervals to be equal[3]. TEMPOS also enhances Timelogic by (optionally) adding a number of axioms to Rhet which allow it to deal with recurrence relations.

As a sample of what TEMPOS will allow us to do, let us examine a simple example.

(1.24) (Define-Time [Time-1] [Time-2] [MaxTime])

This defines three time intervals. Now, assert that Time-1 and Time-2 are separately contained within the interval MaxTime, and further, that Time-1 ends sometime before Time-2 ends.

(1.25) (Assert-Axioms [Time-Contains MaxTime Time-1]
[Time-Contains MaxTime Time-2]
[Time-Finishes-Earlier Time-1 Time-2])

This might be useful, say, if some step in a plan needs to complete earlier than some other step. TEMPOS will allow us to prove that, for instance, there is some time interval disjoint from Time-1 that is contained by Time-2 (since Time-1 finishes earlier than Time-2, this is what we would expect).

[3]This also works in reverse; when Timelogic derives that two intervals unambiguously have relation :E, TEMPOS hooks this derivation and asserts their equality in Rhet.

```
(1.26)  (Prove [and [Time-Skolem [any ?i*t-time
                                      [Time-Disjoint ?i*T-TIME Time-1]]
                                 MaxTime]
                    [Time-Contains-p Time-2 ?i]])
```

Since TEMPOS is fully integrated with Rhet's reasoning mechanisms, one can use it, for instance, in forward chaining axioms. This example might be used to express that two objects cannot be in the same place at the same time.

```
(1.27)  [[Time-Disjoint ?i*t-time ?j*t-time] < [Location ?ob1 ?loc1*T-Location ?i]
                                               [Location ?ob2 ?loc2*T-Location ?j]
                                               ;; True if two objects cannot colocate at
                                               ;; the two locations at same time.
                                               ;; Covers, e.g. impossibility of two
                                               ;; Elephants in the same bathroom
                                               [Location-Overlap? ?ob1 ?ob2 ?loc1 ?loc2]
                                               :forward]
```

In general, reasoning about time intervals in TEMPOS is just like reasoning about any other (non-structured) function term object in Rhet. Since TIMELOGIC supports contextual reasoning, TEMPOS integrates Rhet's context mechanism with TIMELOGIC, allowing, for example, one to reason about time relationships relative to an agent's beliefs.

Chapter 2

Defining an Action Type

Actions are subtypes of the structured type T-ACTION. These have been predefined by RPRS to be a subtype of T-AGENT-OBJECT which is a structured type of one role: R-AGENT which is of type T-ANIMATE. Thus, all actions are expected to have an agent, and (because of the way Rhet's type hierarchies work) the first argument to any Action's constructor function will be the agent. Actions are not expected to have any special relations (that RPRS will handle, anyway), they are interesting insomuch as they are the only valid argument to the Explain-Observations function, that is, these are the only objects that can trigger plan recognition. They are also expected to appear as Steps in Plans (see below).

Here is a trivial example:

```
(2.1)  (DEFINE-ACTION-TYPE 'T-GO-TO-WOODS)
```

2.1 defines a new action type T-GO-TO-WOODS, which is functional (that is, may appear as a constructor function, and further, is unique, that is two instances with the same or equal arguments must also be equal[1]). An example of an instance of this type is `[C-GO-TO-WOODS JOE]`. This might appear, for example, as an argument to Explain-Observations, *e.g.*

```
(2.2)  (EXPLAIN-OBSERVATIONS '([C-GO-TO-WOODS JOE]))
```

might read loosely as "Return a list of all plans that can be constructed in which Joe would go to the woods." Note that, by default, Actions and plans do not have a R-TIME role, though the examples provided with RPRS do, in fact, usually extend T-ACTION to include such a role. We can add such roles to our actions in a straightforward manner. First we construct a new Rhet type, a subtype of T-ACTION (which must be the ancestor of all actions for RPRS to work), which includes any additional roles we wish to consistently utilize in our problem, *e.g.* time:

[1] as defined by Rhet for functional types.

```
(2.3)  (DEFINE-SUBTYPE 'T-MY-ACTION-TYPE 'T-ACT
                       :ROLES '((R-TIME T-TIME)))
```

•

Now we can define an action type that inherits from our extended type by using the optional :PARENT keyword argument, *e.g.*

```
(2.4)  (DEFINE-ACTION-TYPE 'T-DO-SOMETHING-STUPID
                           :PARENT 'T-MY-ACTION-TYPE
                           :ROLES '((R-TIME-LOST T-TIME)))
```

In this example, T-DO-SOMETHING-STUPID could now appear as the parent of some other action; the thing to remember here is that since all types defined with DEFINE-ACTION-TYPE are functional; any time the roles of two instances are equal, then the instances themselves are equal. That is, if we now defined

```
(2.5)  (DEFINE-ACTION-TYPE 'T-DO-SOMETHING-REALLY-STUPID
                           :PARENT 'T-DO-SOMETHING-STUPID
                           :ROLES '((R-DEATHS T-NUMBER)))
```

then `[C-DO-SOMETHING-STUPID JOE NOON-TODAY FOUR-HOURS]` is equal to `[C-DO-SOMETHING-REALLY-STUPID JOE NOON-TODAY FOUR-HOURS FORTY-TWO]` since the first three roles of this second term are equal to (all of) our first instances roles, and both are of type T-DO-SOMETHING-STUPID.

Constructor functions for actions will also typically appear as steps in plans, as described below.

Chapter 3

Defining a Plan Type

Plans are subtypes of the structured type T-PLAN, which is the parent of the two structured types T-RPLAN and T-CPLAN. These have been predefined by RPRS to be a subtype of T-AGENT-OBJECT, as for actions above. Thus, all plans are expected to have an agent, and the first argument to any Plan's constructor function will be the agent. Plans also have a single initialization, which runs the Rhet builtin `[Assert-Relations]`; this is to appropriately instantiate the STEPS that are passed to Define-Rplan-Type or Define-Cplan-Type.

Two functions are provided for defining possible plans; Define-Rplan-Type declares a recognizable plan to RPRS, that is, a plan that can be returned as a result of an Explain-Observations inquiry. Define-Cplan-Type is similar, but may only appear as a step in another Cplan or Rplan[1]. Note that like actions, Cplans are functional; with all the restrictions Rhet places on functional types[2].

As in the example above, it is possible to define normal Rhet types that Cplans and Rplans inherit roles from, **however**, it is important to realize that any relations (*i.e.* step relations) declared on these types will be ignored by RPRS. RPRS depends on the user using it's lisp interface to Rhet in order to intercept and process the relation fields it is interested in. This can be used to advantage, however, *e.g.* if the user wishes to construct a more abstract type that will never itself appear as a step or is desired to be returned as a result by RPRS. The main reason for doing such a thing might be to define abstract steps that are inherited by subtypes without further elaboration. The examples below will demonstrate such usage.

The simplest presentation of how to define an Rplan is via an example. Figure 3.1 shows an instance of an Rplan definition. Here we are defining a plan for making a pasta dish, which will consist of three steps, making the noodles, making the sauce and then boiling the noodles. We constrain the steps so that we must make the noodles before we boil them, and also so that the agent must be italian. Note how the steps defined make

[1]Note that Rplans are restricted to not appear in the step of any other plan. They are *not* functional, as actions and Cplans are.

[2]That is, Rhet will create constructor functions for functional types, and thus two instances with the same roles are inherently equal.

```
(DEFINE-RPLAN-TYPE 'T-make-pasta-dish :PARENT 'T-PREPARE-MEAL
    :STEPS '#[ ((:S-1 . [C-make-noodles [F-AGENT ?SELF]
                                        ?TIME-S-1*T-TIME
                                        ?RESULT*T-NOODLE-DISH])
               (:S-2 . [C-make-sauce [F-AGENT ?SELF]
                                     ?TIME-S-2*T-TIME])
               (:S-3 . [C-boil [F-AGENT ?SELF]
                               ?TIME-S-3*T-TIME
                               ?RESULT*T-NOODLE-DISH])) #]
    :CONSTRAINTS '#[ ([TIME-DURING [F-TIME [S-1 ?SELF]]
                                   [F-TIME ?SELF]]
                      [TIME-DURING [F-TIME [S-2 ?SELF]]
                                   [F-TIME ?SELF]]
                      [TIME-DURING [F-TIME [S-3 ?SELF]]
                                   [F-TIME ?SELF]]
                      [TIME-RELN [F-TIME [S-1 ?SELF]] (:B :M)
                                 [F-TIME [S-3 ?SELF]]]
                      [EQ? [F-RESULT [S-1 ?SELF]]
                           [F-INPUT [S-3 ?SELF]]]
                      [ITALIAN [F-AGENT ?SELF]]) #] )
```

Figure 3.1: Rplan Type Description for T-Make-Pasta-Dish

use of the one role our type has, the (inherited) R-AGENT role. The other variables are simply "placeholders" which allow us to have semantically correct constructor functions[3]. This example shows constraints which use the step accessors on the constructor functions to refer to the roles on the steps. Note that we do not, for instance, have to assert `[EQ? [F-AGENT ?self] [F-AGENT [S-1 ?self]]]` because this was part of the definition of the step.

This particular example is interesting in that a constraint is that the agent must be italian. This means that the agent will be asserted to be italian, if it is not already provably not italian.

[3]Such placeholders are supported by the `[Assert-Relations]` builtin. Note that they are supported only in the constructor function being defined as a relation, not in any constructor functions being used as a term of the relation. That is, `[C-Mumble [C-Foo ?agent*t-agent]]` is illegal, because the placeholder does not occur in the outermost constructor where `[Assert-Relations]` can find it.

Chapter 4

Recognizing Plans from Observed Actions

Once the plan and action hierarchy has been set up, the user need only call Explain-Observations on the list of "observed" actions. The system will then create contexts in which a plan can be instantiated in which the action appears as a step. Since constraints on the plan may cause such instantiation to fail, RPRS will delete the failed context, and try another possibility. Ultimately, it returns a list of contexts so created with the recognized plans (Rplans) it found. The basic algorithm may be paraphrased as follows:

1. For each observed Action find the set of plans that the action appears as a step.
2. For each Cplan in the above set find the set of plans that the plan appears as a step.
3. Repeat 2. above for new Cplans discovered.
4. Eliminate from the set of Rplans those that do not directly or indirectly refer to all of the observed actions.
5. Create a new Rhet user context and instantiate the plan (as a side effect running constraints and initializations of the plan, and instantiating all the steps.
6. If a contradiction has not yet been found, add equalities between the steps of the Rplan and the respective Cplans and Actions that it refers to (recursively for Cplans that are steps).
7. If a contradiction was not found, add the Rplan and context to the result list.

The function Show-Explanation can be used to "pretty-print" the result of Explain-Observations, if desired.

Chapter 5

An Example

This example is taken from the script "RPRS:demo;showoff.lisp", and "RPRS:test;cook-hierarchy-test.lisp"; and is due in part to [Kautz, 1987] and his sample implementation.

First, define some actions...Actions are what we will ask to be explained. They may appear as steps in Cplans or Rplans. In this example we will only deal with Rplans, so all the steps must be Actions. For our own purposes, we want all actions to have a time role, so we will define a new type T-NON-END that is an action with a time. It will be used as the parent of our first "real" action type, T-MAKE-SAUCE.

(5.1) (TSUBTYPE 'T-U 'T-NOODLE-DISH)

(5.2) (DEFINE-SUBTYPE 'T-NON-END 'T-ACTION
:ROLES '((R-TIME T-TIME))) ; all actions have agent roles, btw.

As an effect of Define-Action-Type, we will make T-MAKE-SAUCE a functional structured type. It inherits the roles of all Actions, namely the Agent, and the roles of T-Non-End, it's parent, namely Time. Thus constructor functions should appear as `[C-Make-Sauce Agent Time]`, since the order of the arguments are the roles of the least specific parent first, down the tree until the current type is hit.

(5.3) (DEFINE-ACTION-TYPE 'T-MAKE-SAUCE :PARENT 'T-NON-END)

(5.4) (DEFINE-ACTION-TYPE 'T-MAKE-MARINARA :PARENT 'T-MAKE-SAUCE)

(5.5) (DEFINE-ACTION-TYPE 'T-MAKE-ALFREDO :PARENT 'T-MAKE-SAUCE)

(5.6) (DEFINE-ACTION-TYPE 'T-MAKE-NOODLES :PARENT 'T-NON-END
:ROLES '((R-RESULT T-NOODLE-DISH)))

(5.7) (DEFINE-ACTION-TYPE 'T-MAKE-SPAGHETTI :PARENT 'T-MAKE-NOODLES)

```
(5.8)  (DEFINE-ACTION-TYPE 'T-MAKE-FETTUCINI :PARENT 'T-MAKE-NOODLES)
```

Normally, we might also want to declare that T-Make-Fettucini and T-Make-Spaghetti are disjoint. Similarly for Alfredo and Marinara, or Noodles and Sauce. Since this particular example wouldn't make use of this, we don't bother[1].

```
(5.9)  (DEFINE-ACTION-TYPE 'T-BOIL :PARENT 'T-NON-END
           :ROLES '((R-INPUT T-NOODLE-DISH)))
```

Now set up a recognizable plan hierarchy (note that steps will only be actions defined above, since we are not using Cplans in this example)

```
(5.10) (DEFINE-SUBTYPE 'T-END 'T-RPLAN
           ) ; use this type to capture what our own subtypes need
```

```
(5.11) (DEFINE-SUBTYPE 'T-PREPARE-MEAL 'T-END
           :ROLES '((R-TIME T-TIME)) ;all plans have agent roles, btw.
           :CONSTRAINTS '([INKITCHEN [F-TIME ?SELF] [F-AGENT ?SELF]]))
```

```
(5.12) (DEFINE-RPLAN-TYPE 'T-make-pasta-dish :PARENT 'T-PREPARE-MEAL
           :STEPS '#[ ((:S-1 . [C-make-noodles [F-AGENT ?SELF]
                                               ?TIME-S-1*T-TIME
                                               ?RESULT*T-NOODLE-DISH])
                       (:S-2 . [C-make-sauce [F-AGENT ?SELF]
                                             ?TIME-S-2*T-TIME])
                       (:S-3 . [C-boil [F-AGENT ?SELF]
                                       ?TIME-S-3*T-TIME
                                       ?RESULT*T-NOODLE-DISH])) #]
           :CONSTRAINTS '#[ ([TIME-DURING [F-TIME [S-1 ?SELF]]
                                          [F-TIME ?SELF]]
                             [TIME-DURING [F-TIME [S-2 ?SELF]]
                                          [F-TIME ?SELF]]
                             [TIME-DURING [F-TIME [S-3 ?SELF]]
                                          [F-TIME ?SELF]]
                             [TIME-RELN [F-TIME [S-1 ?SELF]] (:B :M)
                                        [F-TIME [S-3 ?SELF]]]
                             [EQ? [F-RESULT [S-1 ?SELF]]
                                  [F-INPUT [S-3 ?SELF]]]
                             [ITALIAN [F-AGENT ?SELF]]) #] )
```

[1] A more complex example, "language-test" is provided with the system that does need and make use of these disjunction declarations. It's safe to make them in any case.

Now, note the specialization of various steps in the above type definition. For example, we will specialize the first step from a make-noodles to a make-spaghetti, which is a subtype of make-noodles.

```
(5.13) (DEFINE-RPLAN-TYPE 'T-make-spaghetti-marinara
           :PARENT 'T-MAKE-PASTA-DISH
           :STEPS '#[((:S-1 . [C-make-spaghetti [F-AGENT ?SELF]
                                                ?TIME-S-1*T-TIME
                                                ?RESULT*T-NOODLE-DISH])
                      (:S-2 . [C-make-marinara [F-AGENT ?SELF]
                                                ?TIME-S-2*T-TIME]))#])
```

Now try a different specialization.

```
(5.14) (DEFINE-RPLAN-TYPE 'T-make-fettucini-alfredo
           :PARENT 'T-MAKE-PASTA-DISH
           :STEPS '#[((:S-1 . [C-make-fettucini [F-AGENT ?SELF]
                                                ?TIME-S-1*T-TIME
                                                ?RESULT*T-NOODLE-DISH])
                      (:S-2 . [C-make-alfredo [F-AGENT ?SELF]
                                               ?TIME-S-2*T-TIME]))#])

(5.15) (DEFINE-SUBTYPE 'T-MAKE-MEAT-DISH 'T-PREPARE-MEAL)

(5.16) (DEFINE-RPLAN-TYPE 'T-make-chicken-marinara
           :PARENT 'T-make-meat-dish
           :STEPS '((:S-5 . [C-make-marinara [F-AGENT ?SELF]
                                             ?TIME-S-5*T-TIME]))
           :CONSTRAINTS '([TIME-CONTAINS [F-TIME ?SELF]
                                         [F-TIME [S-5 ?SELF]]]))
```

Now, given some appropriately typed instances:

```
(5.17) ;; and some agent instances
       (UTYPE 'T-ANIMATE [JOE] [SALLY])

(5.18) ;; and some result instances
       (UTYPE 'T-NOODLE-DISH [NOODLE-42])
```

At this point, our type hierarchy looks like Figure 5.1.

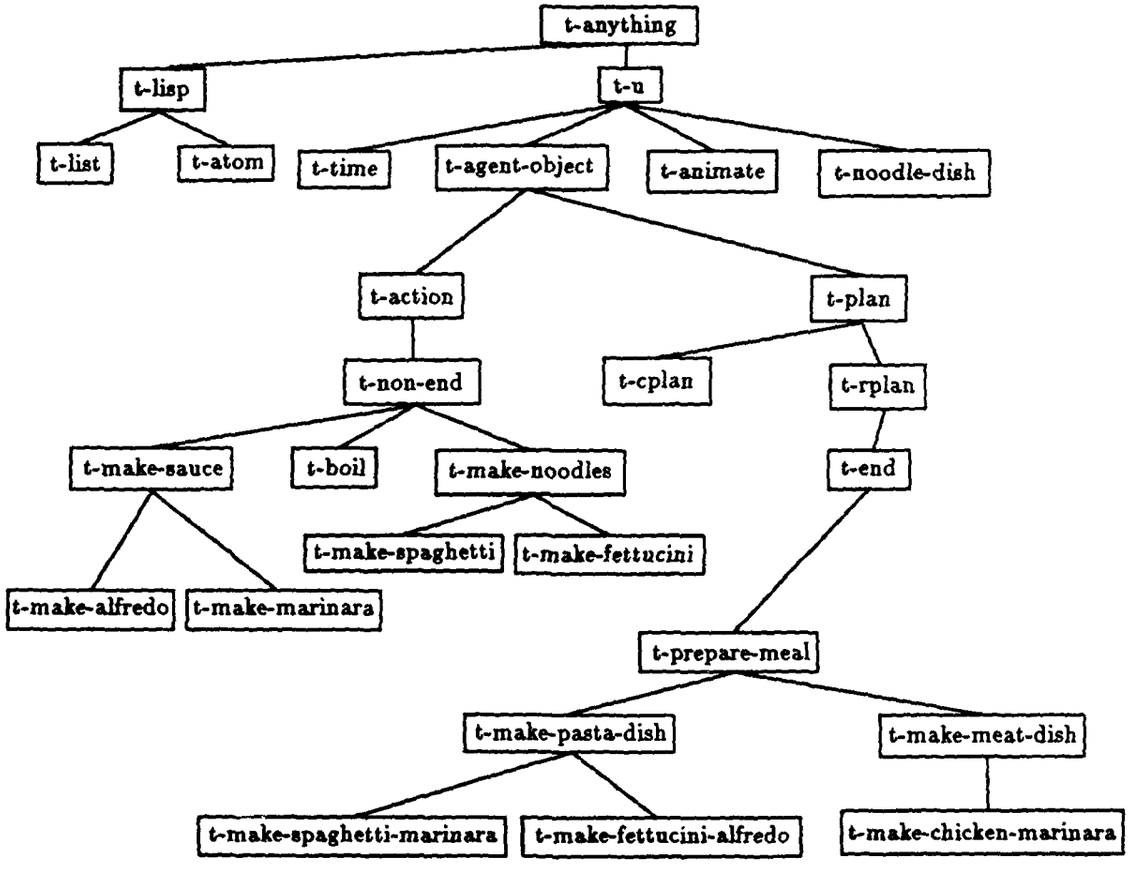

Figure 5.1: Example's Type Hierarchy Expressed as a Tree

Notice that the hierarchy has the constraint that the agent of make-pasta-dish must be Italian[2].

What we are trying to show here is that for some particular observation of Joe making sauce, we will either find four plans (Joe was making Spaghetti Marinara, or Chicken Marinara, or Fettucini Alfredo or some generic Pasta Dish) or one (Chicken Marinara - doesn't involve making pasta for which the agent must be italian) depending on what we know about Joe's heritage. We then run the system on other observations and combinations of observations; the system must find a set of plans, each of which consistently explains all of the observations (as opposed to a set of plans that cover the observations).

Some times are defined, hours last one hour starting at the specified hour. define a time interval that starts between 4 and 5 o'clock and ends between 6 and 7; *etc.*

```
Rhet->
```

```
(5.19)  (LOAD "rprs:test;time-definitions")
```

```
Loading RPRS:TEST;TIME-DEFINITIONS.LISP.NEWEST into package RPRS
Rhet->
```

Create three contexts concerning our agent's status vis the ITALIAN predicate. This is just a direct call to the Rhet lisp interface to create a user context.

```
Rhet->
```

```
(5.20)  (CREATE-UCONTEXT "IS-ITALIAN" "T")
```

```
(◇IS-ITALIAN◇)
Rhet->
```

Note how we will now ask Rhet to assert a fact in a particular user context. This fact is not accessible from the root, thus we can create other user contexts that do not inherit from this one that have incompatible assertions.

```
Rhet->
```

```
(5.21)  (ASSERT-AXIOMS [ITALIAN JOE] :CONTEXT (UCONTEXT "IS-ITALIAN"))
```

```
((SB-IS-ITALIAN::|F-or-T-118623|))
Rhet->
```

Such as this one...

```
Rhet->
```

```
(5.22)  (CREATE-UCONTEXT "IS-not-ITALIAN" "T")
```

[2]Henry had a closed world assumption here about the Italian predicate that we can't deal with due to supporting equality; instead, we run the example three times instead of Henry's two; once when we know the agent IS NOT italian, once when we know he is, and once when we don't comment; we do this in three separate contexts.

```
(◇IS-not-ITALIAN◇)
Rhet->
```

(5.23)
```
(ASSERT-AXIOMS [NOT [ITALIAN JOE]]
                :CONTEXT (UCONTEXT "IS-not-ITALIAN"))
```

```
((|SB-IS-not-ITALIAN|::|F-or-T-118624|))
Rhet->
```

Also create a context where it isn't provable one way or the other. RPRS will make the assertion for us as needed when we try to explain the observations from here.

```
Rhet->
```

(5.24)
```
(CREATE-UCONTEXT "DONT-KNOW" "T")
```

```
(◇DONT-KNOW◇)
Rhet->
```

First observation is make-sauce(obs-sauce), with agent Joe, during time interval beginning between 4 and 5, and ending between 6 and 7.

explain-observation returns a list of pairs of recognized plan instances and the context it created (as a subcontext of the passed user context) in which they were recognized. We map the SHOW-EXPLANATION function over the result which gives us our pretty-printed output.

```
Rhet->
```

(5.25)
```
(MAPCAR #'SHOW-EXPLANATION
        (EXPLAIN-OBSERVATIONS
            [C-MAKE-SAUCE
                JOE
                TIME-STARTS-BETWEEN-4-5-ENDS-BETWEEN-6-7]
            :UCONTEXT "IS-not-ITALIAN"))
```

```
In context: RPRS-Test-Context118772 we found plan [CUR-PLAN118773] of type
:CONSTANT
RPRS::T-MAKE-CHICKEN-MARINARA
(RPRS::R-AGENT [JOE] RPRS::R-TIME [F-TIME CUR-PLAN118773]) ; the slots
 with steps
[C-MAKE-SAUCE JOE TIME-STARTS-BETWEEN-4-5-ENDS-BETWEEN-6-7]     ; step 5
[S-4 CUR-PLAN118773]              ; note that these
[S-3 CUR-PLAN118773]              ; extra unused steps are
[S-2 CUR-PLAN118773]              ; purely an effect of
[S-1 CUR-PLAN118773]              ; the show-explanation fn.

(([[CUR-PLAN118773] "RPRS-Test-Context118772")))
```

Notice that we eliminated the possibility of make-pasta-dish. Therefore the observation had to be of make-marinara. Now lets add the fact that Joe is Italian. This fact does not take a temporal index. Try the observation of make-sauce again.

```
Rhet->

(5.26)  (MAPCAR #'SHOW-EXPLANATION
            (EXPLAIN-OBSERVATIONS
                [C-MAKE-SAUCE
                    JOE
                    TIME-STARTS-BETWEEN-4-5-ENDS-BETWEEN-6-7]
                :UCONTEXT "IS-ITALIAN"))


In context: RPRS-Test-Context118944 we found plan [CUR-PLAN118945] of type
:CONSTANT
RPRS::T-MAKE-CHICKEN-MARINARA
(RPRS::R-AGENT [JOE] RPRS::R-TIME [F-TIME CUR-PLAN118945])
 with steps
[C-MAKE-SAUCE JOE TIME-STARTS-BETWEEN-4-5-ENDS-BETWEEN-6-7]
[S-4 CUR-PLAN118945]
[S-3 CUR-PLAN118945]
[S-2 CUR-PLAN118945]
[S-1 CUR-PLAN118945]

In context: RPRS-Test-Context118894 we found plan [CUR-PLAN118895] of type
:CONSTANT
RPRS::T-MAKE-FETTUCINI-ALFREDO
(RPRS::R-AGENT [JOE] RPRS::R-TIME [F-TIME CUR-PLAN118895])
 with steps
[S-5 CUR-PLAN118895]
[S-4 CUR-PLAN118895]
[C-BOIL JOE [F-TIME [S-3 CUR-PLAN118895]]
                [F-RESULT [C-MAKE-FETTUCINI JOE
                                    [F-TIME [S-1 CUR-PLAN118895]]
                                    [F-RESULT [S-1 CUR-PLAN118895]]]]]
[C-MAKE-SAUCE JOE TIME-STARTS-BETWEEN-4-5-ENDS-BETWEEN-6-7]
[C-MAKE-FETTUCINI JOE
                            [F-TIME [S-1 CUR-PLAN118895]]
                            [F-RESULT [S-1 CUR-PLAN118895]]]

In context: RPRS-Test-Context118844 we found plan [CUR-PLAN118845] of type
:CONSTANT
RPRS::T-MAKE-SPAGHETTI-MARINARA
(RPRS::R-AGENT [JOE] RPRS::R-TIME [F-TIME CUR-PLAN118845])
```

```
 with steps
[S-5 CUR-PLAN118845]
[S-4 CUR-PLAN118845]
[C-BOIL JOE [F-TIME [S-3 CUR-PLAN118845]]
                  [F-RESULT [C-MAKE-SPAGHETTI JOE
                                        [F-TIME [S-1 CUR-PLAN118845]]
                                        [F-RESULT [S-1 CUR-PLAN118845]]]]]
[C-MAKE-SAUCE JOE TIME-STARTS-BETWEEN-4-5-ENDS-BETWEEN-6-7]
[C-MAKE-SPAGHETTI JOE
                              [F-TIME [S-1 CUR-PLAN118845]]
                              [F-RESULT [S-1 CUR-PLAN118845]]]

In context: RPRS-Test-Context118794 we found plan [CUR-PLAN118795] of type
:CONSTANT
RPRS::T-MAKE-PASTA-DISH
(RPRS::R-AGENT [JOE] RPRS::R-TIME [F-TIME CUR-PLAN118795])
 with steps
[S-5 CUR-PLAN118795]
[S-4 CUR-PLAN118795]
[C-BOIL JOE [F-TIME [S-3 CUR-PLAN118795]]
                  [F-RESULT [C-MAKE-NOODLES JOE
                                        [F-TIME [S-1 CUR-PLAN118795]]
                                        [F-RESULT [S-1 CUR-PLAN118795]]]]]
[C-MAKE-SAUCE JOE TIME-STARTS-BETWEEN-4-5-ENDS-BETWEEN-6-7]
[C-MAKE-NOODLES JOE
                            [F-TIME [S-1 CUR-PLAN118795]]
                            [F-RESULT [S-1 CUR-PLAN118795]]]

((([CUR-PLAN118945] "RPRS-Test-Context118944")
  ([CUR-PLAN118895] "RPRS-Test-Context118894")
  ([CUR-PLAN118845] "RPRS-Test-Context118844")
  ([CUR-PLAN118795] "RPRS-Test-Context118794")))
```

This time make-pasta-dish and it's subtypes WERE considered. Note that when we don't know, we will assert the agent (JOE) to be italian in those contexts that he is required to be italian...

```
Rhet->
```

```
(5.27)  (LET ((PLAN-DOT-CONTEXTS
                 (EXPLAIN-OBSERVATIONS
                     [C-MAKE-SAUCE JOE
                         TIME-STARTS-BETWEEN-4-5-ENDS-BETWEEN-6-7]
```

```
                    :UCONTEXT "DONT-KNOW")))
          (MAPC #'(LAMBDA (PLAN-DOT-CONTEXT)
                    (SHOW-EXPLANATION PLAN-DOT-CONTEXT)
                    (FORMAT
                         T "&Italian? ~S"
                         (PROVE
                              [ITALIAN JOE]
                              :CONTEXT
                              (UCONTEXT (CADR PLAN-DOT-CONTEXT)))))
                  PLAN-DOT-CONTEXTS))
```

```
In context: RPRS-Test-Context119115 we found plan [CUR-PLAN119116] of type
:CONSTANT
RPRS::T-MAKE-CHICKEN-MARINARA
(RPRS::R-AGENT [JOE] RPRS::R-TIME [F-TIME CUR-PLAN119116])
 with steps
[C-MAKE-SAUCE JOE TIME-STARTS-BETWEEN-4-5-ENDS-BETWEEN-6-7]
[S-4 CUR-PLAN119116]
[S-3 CUR-PLAN119116]
[S-2 CUR-PLAN119116]
[S-1 CUR-PLAN119116]
Italian? :UNKNOWN

In context: RPRS-Test-Context119065 we found plan [CUR-PLAN119066] of type
:CONSTANT
RPRS::T-MAKE-FETTUCINI-ALFREDO
(RPRS::R-AGENT [JOE] RPRS::R-TIME [F-TIME CUR-PLAN119066])
 with steps
[S-5 CUR-PLAN119066]
[S-4 CUR-PLAN119066]
[C-BOIL JOE [F-TIME [S-3 CUR-PLAN119066]]
                  [F-RESULT [C-MAKE-FETTUCINI JOE
                                    [F-TIME [S-1 CUR-PLAN119066]]
                                    [F-RESULT [S-1 CUR-PLAN119066]]]]]
[C-MAKE-SAUCE JOE TIME-STARTS-BETWEEN-4-5-ENDS-BETWEEN-6-7]
[C-MAKE-FETTUCINI JOE [F-TIME [S-1 CUR-PLAN119066]]
                                  [F-RESULT [S-1 CUR-PLAN119066]]]
Italian? [ITALIAN JOE]

In context: RPRS-Test-Context119015 we found plan [CUR-PLAN119016] of type
:CONSTANT
RPRS::T-MAKE-SPAGHETTI-MARINARA
(RPRS::R-AGENT [JOE] RPRS::R-TIME [F-TIME CUR-PLAN119016])
```

```
 with steps
[S-5 CUR-PLAN119016]
[S-4 CUR-PLAN119016]
[C-BOIL JOE [F-TIME [S-3 CUR-PLAN119016]]
                  [F-RESULT [C-MAKE-SPAGHETTI JOE
                                    [F-TIME [S-1 CUR-PLAN119016]]
                                    [F-RESULT [S-1 CUR-PLAN119016]]]]]
[C-MAKE-SAUCE JOE TIME-STARTS-BETWEEN-4-5-ENDS-BETWEEN-6-7]
[C-MAKE-SPAGHETTI JOE [F-TIME [S-1 CUR-PLAN119016]]
                                  [F-RESULT [S-1 CUR-PLAN119016]]]
Italian? [ITALIAN JOE]

In context: RPRS-Test-Context118965 we found plan [CUR-PLAN118966] of type
:CONSTANT
RPRS::T-MAKE-PASTA-DISH
(RPRS::R-AGENT [JOE] RPRS::R-TIME [F-TIME CUR-PLAN118966])
 with steps
[S-5 CUR-PLAN118966]
[S-4 CUR-PLAN118966]
[C-BOIL JOE [F-TIME [S-3 CUR-PLAN118966]]
                  [F-RESULT [C-MAKE-NOODLES JOE
                                    [F-TIME [S-1 CUR-PLAN118966]]
                                    [F-RESULT [S-1 CUR-PLAN118966]]]]]
[C-MAKE-SAUCE JOE TIME-STARTS-BETWEEN-4-5-ENDS-BETWEEN-6-7]
[C-MAKE-NOODLES JOE [F-TIME [S-1 CUR-PLAN118966]]
                                [F-RESULT [S-1 CUR-PLAN118966]]]
Italian? [ITALIAN JOE]

((([CUR-PLAN119116] "RPRS-Test-Context119115")
   ([CUR-PLAN119066] "RPRS-Test-Context119065")
   ([CUR-PLAN119016] "RPRS-Test-Context119015")
  ([CUR-PLAN118966] "RPRS-Test-Context118965")))
```

The second observation is making noodles.

```
Rhet->

(5.28)  (MAPCAR #'SHOW-EXPLANATION
            (EXPLAIN-OBSERVATIONS
                [C-MAKE-NOODLES
                     JOE
                     TIME-STARTS-BETWEEN-6-8-ENDS-BETWEEN-7-9
                     NOODLE-42]
                :UCONTEXT "IS-ITALIAN"))
```

```
In context: RPRS-Test-Context119266 we found plan [CUR-PLAN119267] of type
:CONSTANT
RPRS::T-MAKE-FETTUCINI-ALFREDO
(RPRS::R-AGENT [JOE] RPRS::R-TIME [F-TIME CUR-PLAN119267])
 with steps
[S-5 CUR-PLAN119267]
[S-4 CUR-PLAN119267]
[C-BOIL JOE [F-TIME [S-3 CUR-PLAN119267]] NOODLE-42]
[C-MAKE-ALFREDO JOE [F-TIME [S-2 CUR-PLAN119267]]]
[C-MAKE-NOODLES JOE
    TIME-STARTS-BETWEEN-6-8-ENDS-BETWEEN-7-9 NOODLE-42]

In context: RPRS-Test-Context119207 we found plan [CUR-PLAN119208] of type
:CONSTANT
RPRS::T-MAKE-SPAGHETTI-MARINARA
(RPRS::R-AGENT [JOE] RPRS::R-TIME [F-TIME CUR-PLAN119208])
 with steps
[S-5 CUR-PLAN119208]
[S-4 CUR-PLAN119208]
[C-BOIL JOE [F-TIME [S-3 CUR-PLAN119208]] NOODLE-42]
[C-MAKE-MARINARA JOE [F-TIME [S-2 CUR-PLAN119208]]]
[C-MAKE-NOODLES JOE
    TIME-STARTS-BETWEEN-6-8-ENDS-BETWEEN-7-9 NOODLE-42]

In context: RPRS-Test-Context119157 we found plan [CUR-PLAN119158] of type
:CONSTANT
RPRS::T-MAKE-PASTA-DISH
(RPRS::R-AGENT [JOE] RPRS::R-TIME [F-TIME CUR-PLAN119158])
 with steps
[S-5 CUR-PLAN119158]
[S-4 CUR-PLAN119158]
[C-BOIL JOE [F-TIME [S-3 CUR-PLAN119158]] NOODLE-42]
[C-MAKE-SAUCE JOE [F-TIME [S-2 CUR-PLAN119158]]]
[C-MAKE-NOODLES JOE
    TIME-STARTS-BETWEEN-6-8-ENDS-BETWEEN-7-9 NOODLE-42]

((([CUR-PLAN119267] "RPRS-Test-Context119266")
  ([CUR-PLAN119208] "RPRS-Test-Context119207")
  ([CUR-PLAN119158] "RPRS-Test-Context119157")))
```

We can "merge" these together: they can be steps of the same action, a make-pasta-dish (though we have to rerun the explanation generator)...

```
Rhet->

(5.29)  (MAPCAR #'SHOW-EXPLANATION
            (EXPLAIN-OBSERVATIONS
                [C-MAKE-SAUCE
                    JOE
                    TIME-STARTS-BETWEEN-4-5-ENDS-BETWEEN-6-7]
                    [C-MAKE-NOODLES
                        JOE
                        TIME-STARTS-BETWEEN-6-8-ENDS-BETWEEN-7-9
                        NOODLE-42]
                :UCONTEXT "IS-ITALIAN"))


In context: RPRS-Test-Context119424 we found plan [CUR-PLAN119425] of type
:CONSTANT
RPRS::T-MAKE-FETTUCINI-ALFREDO
(RPRS::R-AGENT [JOE] RPRS::R-TIME [F-TIME CUR-PLAN119425])
 with steps
[S-5 CUR-PLAN119425]
[S-4 CUR-PLAN119425]
[C-BOIL JOE [F-TIME [S-3 CUR-PLAN119425]] NOODLE-42]
[C-MAKE-SAUCE JOE TIME-STARTS-BETWEEN-4-5-ENDS-BETWEEN-6-7]
[C-MAKE-NOODLES JOE
     TIME-STARTS-BETWEEN-6-8-ENDS-BETWEEN-7-9 NOODLE-42]

In context: RPRS-Test-Context119370 we found plan [CUR-PLAN119371] of type
:CONSTANT
RPRS::T-MAKE-SPAGHETTI-MARINARA
(RPRS::R-AGENT [JOE] RPRS::R-TIME [F-TIME CUR-PLAN119371])
 with steps
[S-5 CUR-PLAN119371]
[S-4 CUR-PLAN119371]
[C-BOIL JOE [F-TIME [S-3 CUR-PLAN119371]] NOODLE-42]
[C-MAKE-SAUCE JOE TIME-STARTS-BETWEEN-4-5-ENDS-BETWEEN-6-7]
[C-MAKE-NOODLES JOE
     TIME-STARTS-BETWEEN-6-8-ENDS-BETWEEN-7-9 NOODLE-42]

In context: RPRS-Test-Context119316 we found plan [CUR-PLAN119317] of type
:CONSTANT
RPRS::T-MAKE-PASTA-DISH
(RPRS::R-AGENT [JOE] RPRS::R-TIME [F-TIME CUR-PLAN119317])
 with steps
[S-5 CUR-PLAN119317]
```

```
[S-4 CUR-PLAN119317]
[C-BOIL JOE [F-TIME [S-3 CUR-PLAN119317]] NOODLE-42]
[C-MAKE-SAUCE JOE TIME-STARTS-BETWEEN-4-5-ENDS-BETWEEN-6-7]
[C-MAKE-NOODLES JOE
     TIME-STARTS-BETWEEN-6-8-ENDS-BETWEEN-7-9 NOODLE-42]

((([CUR-PLAN119425] "RPRS-Test-Context119424")
  ([CUR-PLAN119371] "RPRS-Test-Context119370")
  ([CUR-PLAN119317] "RPRS-Test-Context119316")))
```

Now consider an observation of make-noodles with a different agent. Constants like agent are considered unique names (via the ADD-INEQ function), and thus all are unequal.

```
Rhet->

(5.30)  (ASSERT-AXIOMS [ITALIAN SALLY])

((SBMB-T::|F-or-T-119478|))
Rhet->

(5.31)  (ADD-INEQ [JOE] [SALLY])

(T)
Rhet->

(5.32)  (MAPCAR #'SHOW-EXPLANATION
            (EXPLAIN-OBSERVATIONS
                [C-MAKE-NOODLES
                    SALLY
                    TIME-STARTS-BETWEEN-6-8-ENDS-BETWEEN-7-9
                    NOODLE-42]))


In context: RPRS-Test-Context119589 we found plan [CUR-PLAN119590] of type
:CONSTANT
RPRS::T-MAKE-FETTUCINI-ALFREDO
(RPRS::R-AGENT [SALLY] RPRS::R-TIME [F-TIME CUR-PLAN119590])
 with steps
[S-5 CUR-PLAN119590]
[S-4 CUR-PLAN119590]
[C-BOIL SALLY [F-TIME [S-3 CUR-PLAN119590]] NOODLE-42]
[C-MAKE-ALFREDO SALLY [F-TIME [S-2 CUR-PLAN119590]]]
[C-MAKE-NOODLES SALLY
     TIME-STARTS-BETWEEN-6-8-ENDS-BETWEEN-7-9 NOODLE-42]
```

```
In context: RPRS-Test-Context119530 we found plan [CUR-PLAN119531] of type
:CONSTANT
RPRS::T-MAKE-SPAGHETTI-MARINARA
(RPRS::R-AGENT [SALLY] RPRS::R-TIME [F-TIME CUR-PLAN119531])
 with steps
[S-5 CUR-PLAN119531]
[S-4 CUR-PLAN119531]
[C-BOIL SALLY [F-TIME [S-3 CUR-PLAN119531]] NOODLE-42]
[C-MAKE-MARINARA SALLY [F-TIME [S-2 CUR-PLAN119531]]]
[C-MAKE-NOODLES SALLY
     TIME-STARTS-BETWEEN-6-8-ENDS-BETWEEN-7-9 NOODLE-42]

In context: RPRS-Test-Context119480 we found plan [CUR-PLAN119481] of type
:CONSTANT
RPRS::T-MAKE-PASTA-DISH
(RPRS::R-AGENT [SALLY] RPRS::R-TIME [F-TIME CUR-PLAN119481])
 with steps
[S-5 CUR-PLAN119481]
[S-4 CUR-PLAN119481]
[C-BOIL SALLY [F-TIME [S-3 CUR-PLAN119481]] NOODLE-42]
[C-MAKE-SAUCE SALLY [F-TIME [S-2 CUR-PLAN119481]]]
[C-MAKE-NOODLES SALLY
     TIME-STARTS-BETWEEN-6-8-ENDS-BETWEEN-7-9 NOODLE-42]

((([CUR-PLAN119590] "RPRS-Test-Context119589")
  ([CUR-PLAN119531] "RPRS-Test-Context119530")
  ([CUR-PLAN119481] "RPRS-Test-Context119480")))
```

Try to match this with the original make-sauce. It will fail, because agent roles differ[3].

```
Rhet->

(5.33)  (MAPCAR #'SHOW-EXPLANATION
            (EXPLAIN-OBSERVATIONS
                [C-MAKE-SAUCE JOE
                    TIME-STARTS-BETWEEN-4-5-ENDS-BETWEEN-6-7]
                [C-MAKE-NOODLES SALLY
                    TIME-STARTS-BETWEEN-6-8-ENDS-BETWEEN-7-9
                    NOODLE-42]
                :UCONTEXT "IS-ITALIAN"))
```

[3] Actually, while Henry could do this, we have to rerun the explanation proof.

```
(NIL)
Rhet->
```

Lets check out the temporal constraints, now. We'll observe a boiling event, but the time we be BEFORE the make-noodles event. This conflict will prevent a match.

```
Rhet->

(5.34)  (MAPCAR #'SHOW-EXPLANATION
            (EXPLAIN-OBSERVATIONS
                [C-BOIL SALLY HOUR-1 NOODLE-42]
                [C-MAKE-NOODLES
                    SALLY
                    TIME-STARTS-BETWEEN-6-8-ENDS-BETWEEN-7-9
                    NOODLE-42]))

(NIL)
Rhet->
```

Now lets find a LATER boiling event. It should match okay.

```
Rhet->

(5.35)  (MAPCAR #'SHOW-EXPLANATION
            (EXPLAIN-OBSERVATIONS
                [C-BOIL SALLY
                        TIME-STARTS-BETWEEN-9-10-ENDS-BETWEEN-11-12
                        NOODLE-42]
                [C-MAKE-NOODLES
                    SALLY
                    TIME-STARTS-BETWEEN-6-8-ENDS-BETWEEN-7-9
                    NOODLE-42]))


In context: RPRS-Test-Context120053 we found plan [CUR-PLAN120054] of type
:CONSTANT
RPRS::T-MAKE-FETTUCINI-ALFREDO
(RPRS::R-AGENT [SALLY] RPRS::R-TIME [F-TIME CUR-PLAN120054])
 with steps
[S-5 CUR-PLAN120054]
[S-4 CUR-PLAN120054]
[C-BOIL SALLY TIME-STARTS-BETWEEN-9-10-ENDS-BETWEEN-11-12 NOODLE-42]
[C-MAKE-ALFREDO SALLY [F-TIME [S-2 CUR-PLAN120054]]]
[C-MAKE-NOODLES SALLY
     TIME-STARTS-BETWEEN-6-8-ENDS-BETWEEN-7-9 NOODLE-42]

In context: RPRS-Test-Context120000 we found plan [CUR-PLAN120001] of type
```

```
:CONSTANT
RPRS::T-MAKE-SPAGHETTI-MARINARA
(RPRS::R-AGENT [SALLY] RPRS::R-TIME [F-TIME CUR-PLAN120001])
 with steps
[S-5 CUR-PLAN120001]
[S-4 CUR-PLAN120001]
[C-BOIL SALLY TIME-STARTS-BETWEEN-9-10-ENDS-BETWEEN-11-12 NOODLE-42]
[C-MAKE-MARINARA SALLY [F-TIME [S-2 CUR-PLAN120001]]]
[C-MAKE-NOODLES SALLY
     TIME-STARTS-BETWEEN-6-8-ENDS-BETWEEN-7-9 NOODLE-42]

In context: RPRS-Test-Context119946 we found plan [CUR-PLAN119947] of type
:CONSTANT
RPRS::T-MAKE-PASTA-DISH
(RPRS::R-AGENT [SALLY] RPRS::R-TIME [F-TIME CUR-PLAN119947])
 with steps
[S-5 CUR-PLAN119947]
[S-4 CUR-PLAN119947]
[C-BOIL SALLY TIME-STARTS-BETWEEN-9-10-ENDS-BETWEEN-11-12 NOODLE-42]
[C-MAKE-SAUCE SALLY [F-TIME [S-2 CUR-PLAN119947]]]
[C-MAKE-NOODLES SALLY
     TIME-STARTS-BETWEEN-6-8-ENDS-BETWEEN-7-9 NOODLE-42]

((([CUR-PLAN120054] "RPRS-Test-Context120053")
 ([CUR-PLAN120001] "RPRS-Test-Context120000")
 ([CUR-PLAN119947] "RPRS-Test-Context119946")))
```

Chapter 6

RPRS Lisp Interface to Rhet

This chapter intends to show how RPRS uses Rhet (and TEMPOS), and serve as a guide to the user building other systems that use Rhet as their KR engine.

6.1 Overall Design

RPRS uses Rhet in two different distinct ways. First, it asserts to Rhet an association between plan steps of a plan type and action types. RPRS supplies a simple PROLOG like Horn clause program to derive a set of Rplans that each completely cover the presented observations. Second it uses Rhet to (attempt) to instantiate these Rplans, each in their own context, including the user's constraints, initializations, and steps. Should an error occur at this time, RPRS tosses the Rplan as inconsistent. Normally a plan would be uninstantiable because there is an inconsistency between the constraints of the structured type and the actual observation. For instance, in the example in the last chapter, the Agent object of all steps had to be identical, and trying to recognize observations by two agents caused no Rplans to be found.

One reason for separating recognition into two phases[1] is to maximize the amount of work that does NOT involve Rhet's equality system (which is what maintains most of the information about structured types, *e.g.* accessor slot equivalence). Adding equalities is a very inefficient operation in Rhet, so it should be avoided whenever possible[2]

In the following sections, various portions of the RPRS code are presented and discussed.

6.2 Initialization and Mode Setup

[1] other than pedagogical ones

[2] Note that looking up equalities is very efficient; the price is paid at add time on the assumption that proof steps that will check the equality occur much more often than adding new information.

```
(DEFUN RESET-RPRS ()
  "Reset the RPRS system to the just-loaded and initialized condition."
  (RESET-RHETORICAL)
  (DEFINE-REP-RELATION :STEPS :INHERIT-TYPE :MERGE
                       :FN-DEFINITION-HOOK 'STEP-RELATION-DEFINITION-HOOK)
  (SET-CONTRADICTION-MODE :THROW 'RPRS-CONTRADICTION-FOUND)
  (TEMPOS::RESET-TEMPOS :TT-AXIOMS-P T :AUTO-REFERENCE :ON)
  (SETQ RHET-TERMS:*INHIBIT-MAXTYPE-WARNINGS* T)        ;yeah, yeah.
```

Figure 6.1: RPRS's Reset-RPRS function - initialization

Figure 6.1 and 6.2 display the RPRS reset function. In figure 6.1 we reset the underlying reasoning systems (both Rhet and Tempos), as well as defining a definition hook for the :STEPS relations type, which appears in figure 6.3.

Figure 6.2 defines useful types for RPRS to use for representing plans and actions. Note the usage of the Initializations option on the T-PLAN type; all plans will have a steps relation, each element of the relation list will have a CAR which is an accessor's name for the step, and a CDR which is a constructor function; the step itself.

The hook itself (in figure 6.3) then uses this notion of a step to find the function type of the accessor, if it exists[3]. In either case, the hook tells Rhet that the accessor when presented with an argument of the type we are defining will result in the type of the constructor function in the step itself.

The TDisjoint and TXSubtype calls make Rhet more efficient, since obvious bad bindings for a variable are discarded sooner.

6.3 RPRS Type Definition Functions

Defining new actions, as in figure 6.4, is as simple as translating the call into the appropriate one for Rhet. Defining new plans, as in figures 6.5[4] and 6.6, is more complicated. Here, RPRS makes various Rhet assertions about the structure of the plans. In particular, we first declare the new type to Rhet using a TSubType call. Then we use Rhet's structured type functions (either Define-Functional-Subtype or Define-Subtype) to instantiate the plan as a Rhet structured type with a STEPS relation of the passed steps. Last we process each step in order to make an assertion of the form:

(6.1) `[HAS-STEP :CPLAN ?plan-type ?step-type ?step-constructor-fn]`

[3] If not, it can define a maxtype, which is used by the parser to constrain the legal types of arguments in a function term.

[4] Define-Rplan-Type is similar.

```
;; give T-AGENT-OBJECT a REP name later,
;; but need to maxtype F-AGENT first.
(TSUBTYPE 'T-U 'T-AGENT-OBJECT)
(DEFINE-SUBTYPE 'T-ANIMATE 'T-U)
(DECLARE-FN-TYPE 'F-AGENT '(T-AGENT-OBJECT T-ANIMATE)) ; set maxtype
(DEFINE-SUBTYPE 'T-AGENT-OBJECT 'T-U
                :ROLES '((R-AGENT T-ANIMATE)))
(TDISJOINT 'T-ANIMATE 'T-AGENT-OBJECT 'T-TIME)
;; plans must be a subtype of this. Necessary slots
;; common to all plans defined here.
(DEFINE-SUBTYPE 'T-PLAN 'T-AGENT-OBJECT
  ;; force them to provide their own accessors, this is a bit more
  ;; general, and besides, we need to do declare-fn-type. (via hook)
  :INITIALIZATIONS (LIST (CONS-RHET-FORM
                           'RLLIB::ASSERT-RELATIONS
                           (RHET-TERMS:CREATE-RVARIABLE "?SELF")
                           :STEPS
                           :T)))
;; actions (steps in a plan) must be a subtype of this. Necessary slots
;; common to all actions defined here.
(DEFINE-SUBTYPE 'T-ACTION 'T-AGENT-OBJECT)
(TDISJOINT 'T-ACTION 'T-PLAN) ; plans are not actions, and vice-versa
;; Plans that can be construed as a complete plan are subtypes of this.
;; See thesis for information about "END" types. (Henry makes END an ab-
;; straction for certain kinds of plans, RPRS makes it a subtype relation-
;; ship only. This also prevents going thru each abstraction separately.)
(DEFINE-SUBTYPE 'T-RPLAN 'T-PLAN)
;; for non-end (recognizable) plans, they inherit from this.
(DEFINE-SUBTYPE 'T-CPLAN 'T-PLAN)
;; note that these types partition the plan type, either you are a
;; recognizable plan, or you are not.
(TXSUBTYPE 'T-PLAN 'T-CPLAN 'T-RPLAN)
(LOAD "rprs:code;rhet-code.lisp"))
```

Figure 6.2: RPRS's Reset-RPRS function - type hierarchy

```
(DEFUN STEP-RELATION-DEFINITION-HOOK (TYPE STEPS)
  "Called when define-(functional-)subtype is about to reparse
constraints/relations/initializations.
This lets step accessors be properly declared first."
  (MAPC #'(LAMBDA (STEP)
            (UNLESS (LOOK-UP-FN-TYPE (CAR STEP))
              ;; never before defined;
              ;; this will set an appropriate maxtype.
              (DECLARE-FN-TYPE (CAR STEP) (LIST 'T-PLAN 'T-AGENT-OBJECT))
            (ADD-FN-TYPE (CAR STEP)
                         (LIST TYPE
                               (LET ((RHET-TERMS::*BE-PERMISSIVE* T))
                                 (DECLARE (SPECIAL
                                            RHET-TERMS::*BE-PERMISSIVE*))
                                 (GET-TYPE-OBJECT (CDR STEP))))))
        STEPS))
```

Figure 6.3: Hook to describe step usage to define-subtype

```
(DEFUN DEFINE-ACTION-TYPE (TYPE &REST ARGS
                                &KEY (PARENT 'T-ACTION) &ALLOW-OTHER-KEYS)
  "Defines a RPRS compatible Action type, which may appear as a constructor
function in a step.

Warning: be sure to define these BEFORE use, otherwise types won't get
updated properly.
(and RPRS will not find any applicable plans with steps.)"

  (APPLY #'DEFINE-FUNCTIONAL-SUBTYPE (LIST* TYPE PARENT ARGS)))
```

Figure 6.4: Defining an Action Type

```
(DEFUN DEFINE-CPLAN-TYPE (TYPE &REST ARGS &KEY (PARENT 'T-CPLAN) STEPS
                                              &ALLOW-OTHER-KEYS)
  "Defines a RPRS compatible Plan type, which may NOT be returned by the
recognizer as a 'complete' plan, rather it is a Constituent PLAN"

  (DEFINE-PLAN-TYPE-COMMON TYPE PARENT :CPLAN STEPS ARGS T))
```

Figure 6.5: Cplan Type Definition

which will be used later by the explain-observations function to find what plans have a particular observed step. Note that this function handles steps that don't have a CAR of the accessor, and so generates the default one that Assert-Relations would expect. While this is contrary to the usage declared in our Reset-RPRS function, it is more robust; it allows us to change how we handle this later[5].

6.4 The Explanation Generator

6.4.1 Finding Relevant Plan Types to Instantiate

This is the guts of the frob. Figure 6.7 shows us taking our event-list and finding a series of proofs based on the predicate [Cover-Observations], presented in figure 6.10, which is the entry point into a Rhet BC program elaborated on further in the various figures.

6.4.2 Instantiation of Plans

Now we have a list of proofs, which are essentially just a list of plans each of which explain all of the observations, but are not necessarily internally consistent with the actual observations.

In order to find any possible inconsistency we must instantiate the actual plans, which will cause the definitions of types, the relations, constraints between roles, *etc.* to be expanded. Figure 6.16 shows the remainder of the Explain-Observations function which does this instantiation. The interesting point is that we set up a catch for inconsistencies Rhet detects, and do each instantiation in a gensym'd child context... That way the inconsistencies are removed when we destroy this temporary context. Otherwise the context is left and returned as part of the result of the explain.

[5]The only dependency that forces us to use explicit accessors is our hook function into Define-Subtype. If the hook were expanded to include the relative offset into the list of relations...

```
(DEFUN DEFINE-PLAN-TYPE-COMMON (TYPE PARENT-TYPE CODE STEPS ARGS
                                &OPTIONAL FUNCTIONAL)
    (COND
      (STEPS
       ;; define-subtype would do the tsubtype call, but we want to define
       ;; appropriate fn types for our steps first.
       (TSUBTYPE PARENT-TYPE TYPE)
       (APPLY (IF FUNCTIONAL #'DEFINE-FUNCTIONAL-SUBTYPE #'DEFINE-SUBTYPE)
              TYPE PARENT-TYPE
              :RELATIONS `((:STEPS ,@STEPS)
                           ,@(GRAB-KEY :RELATIONS ARGS)) ARGS)
       ;; invert the steps
       ;; (we could handle, but don't, having step relations inside of ARGS)
       (LET ((NUMBER 0))
         ;; this picks up inherited steps so they get properly inverted.
         (DOLIST (STEP (RHET-TERMS:GET-RELATIONS :STEPS TYPE))
           (INCF NUMBER)
           (LET ((STEP-TYPE (GET-TYPE-OBJECT (IF (CONSP STEP)
                                                 ;; Has accessor
                                                 (CDR STEP)
                                                 STEP)))
                 ;; we don't currently handle accessor string for
                 ;; assert-relations, only :t or default.
                 (CURRENT-ACCESSOR (IF (CONSP STEP)
                                       (CAR STEP) ; accessor
                                       (INTERN    ; default
                                         (FORMAT NIL "STEPS-%d" NUMBER)
                                         'KEYWORD))))
             (LET ((AXIOM (CONS-RHET-AXIOM
                           (CONS-RHET-FORM
                              :HAS-STEP CODE
                              (INTERN (STRING TYPE) 'KEYWORD)
                              (INTERN (STRING STEP-TYPE) 'KEYWORD)
                              CURRENT-ACCESSOR))))
               (SETF (B-AX:BASIC-AXIOM-INDEX AXIOM)
                     "INDEX-<RPRS-ASSERTION")
               (ASSERT-AXIOMS AXIOM))))))
      (T
       (APPLY (IF FUNCTIONAL #'DEFINE-FUNCTIONAL-SUBTYPE #'DEFINE-SUBTYPE)
              TYPE PARENT-TYPE ARGS))))
```

Figure 6.6: Common Plan Type Definition

```
(DEFUN EXPLAIN-OBSERVATIONS (&REST EVENT-LIST)
  "Return a list of lists whose cars are a possible Plan that describe all
of EVENT-LIST, and whose cdr is the context this instance is instantiated
in. Each EVENT object is an instance of a subtype of T-Action.

Note that one can make following calls to Explain-Observations passing one
of these returned contexts; this would constrain the world to be
consistent with the previous discovered plan."
  (DECLARE (ARGLIST (&REST EVENTS &OPTIONAL (UCONTEXT "T"))))
  ;; try to find entries

  ;; this step will depend on the particular algorithm. Now, assume they
  ;; must be related

  (LET* ((UCONTEXT (GRAB-KEY :UCONTEXT EVENT-LIST "T"))
         (EVENT-LIST (TRUNCATE-KEYWORDS EVENT-LIST))
         (PLAN-LIST (RHET-TERMS:CREATE-RVARIABLE
                      "?plan-list" RHET-TERMS:*T-LIST-ITYPE-STRUCT*))
         ;; each value is
         ;; [has-step-recursive event-type plan-type (plan-type step)]
         CANDIDATES
         RETURN-VALUE)
    (PROVE (CONS-RHET-FORM 'COVER-OBSERVATIONS EVENT-LIST PLAN-LIST)
           :MODE :SIMPLE)       ;side effect changes plan-list
    (SETQ CANDIDATES (E-UNIFY:CRUNCH-VARS-IN-ARBFORM
                       (E-UNIFY:GET-VAR-REFERENCE PLAN-LIST)))

    ;; now we must instantiate each possibility so Rhet will calculate
    ;; constraints. Set up handlers for certain kinds of errors Rhet may
    ;; generate. 'rprs-contradiction-found is thrown by rhet thanks to the
    ;; set-contradiction-mode in reset-rprs.
    ;;
    ;; Catch them here and then we know that particular plan is impossible.
    (FORMAT T "Candidates for matching: ~S" CANDIDATES)
```

Figure 6.7: Generating Explanations - Doing Proofs in Rhet

```
#[              ;keep ?step-type from being detected as "different".
(DEFRHETPRED COVER-INITIAL-OBSERVATION (?STEP*T-U ?PLAN-LIST*T-LIST)
  <RPRS [TYPE? ?STEP ?STEP-TYPE*T-ANYTHING]
        [SETVALUE ?PLAN-LIST-PROTOTYPE*T-LIST
                  (PROVE-ALL [HAS-STEP-RECURSIVE ?STEP-TYPE
                                                 ?STEP
                                                 ?RPLAN-TYPE*T-ATOM
                                                 ?ASSERT-LIST*T-LIST]
                             :MODE :SIMPLE)]
        [PROOF-LIST-TO-STEP-LIST ?PLAN-LIST-PROTOTYPE ?PLAN-LIST])
#]
```

Figure 6.8: Rhet Code for Finding Plan Types - Cover-Initial-Observation

```
(ASSERT-AXIOMS
  [ [COVER-OBSERVATION ?STEP
                       ((?OLD-PLAN-TYPE*T-LISP . ?OLD-PLAN-STEPS)
                        . ?MORE-OLD-PLANS)
                       ?NEW-PLAN-LIST*T-LIST]
   <RPRS [TYPE? ?STEP ?STEP-TYPE*T-ANYTHING]
         [SETVALUE ?NEW-PLAN-PROTOTYPES*T-LIST
                   (PROVE-ALL [HAS-STEP-RECURSIVE ?STEP-TYPE
                                                  ?STEP
                                                  ?OLD-PLAN-TYPE
                                                  ?ASSERTIONS*T-LIST]
                              :MODE :SIMPLE)]
         [PROOF-LIST-TO-STEP-LIST ?NEW-PLAN-PROTOTYPES
                                  ?NEW-PLAN-LIST-1*T-LIST]
         [MERGE-PLAN-INSTANCES ?OLD-PLAN-TYPE
                               ?OLD-PLAN-STEPS
                               ?NEW-PLAN-LIST-1
                               ?MERGED-PLAN-LIST*T-LIST]
         [COVER-OBSERVATION ?STEP ?MORE-OLD-PLANS ?NEW-PLAN-LIST-2*T-LIST]
         [APPEND ?MERGED-PLAN-LIST ?NEW-PLAN-LIST-2 ?NEW-PLAN-LIST]]

  [[COVER-OBSERVATION ?STEP NIL NIL] <RPRS])
```

Figure 6.9: Rhet Code for Finding Plan Types - Cover-Observation

```
#[
(DEFRHETPRED COVER-OBSERVATIONS ((?STEP*T-U . ?STEP-LIST*T-LIST)
                                 ?NEW-PLAN-LIST*T-LIST)
   <RPRS [COND ; use COND for example purposes
          ([NULL ?STEP-LIST]
           [COVER-INITIAL-OBSERVATION ?STEP ?NEW-PLAN-LIST])
          ([WIN]
           [COVER-OBSERVATIONS ?STEP-LIST ?NEW-PLAN-LIST-1*T-LIST]
           [COVER-OBSERVATION ?STEP ?NEW-PLAN-LIST-1 ?NEW-PLAN-LIST])])
#]
```

Figure 6.10: Rhet Code for Finding Plan Types - Cover-Observations

```
(ASSERT-AXIOMS
  [[PROOF-LIST-TO-STEP-LIST ([HAS-STEP-RECURSIVE ?STEP-TYPE*T-LISP
                                                 ?STEP*T-U
                                                 ?RPLAN-TYPE*T-LISP
                                                 ?ASSERT-LIST*T-LIST]
                             . ?MORE-PROOFS)
                            (?ASSERT-LIST . ?MORE-PLAN-LISTS)]
   <RPRS [PROOF-LIST-TO-STEP-LIST ?MORE-PROOFS ?MORE-PLAN-LISTS]]
  [[PROOF-LIST-TO-STEP-LIST NIL NIL] <RPRS])
```

Figure 6.11: Rhet Support Code for Parsing Plan Proofs - Proof-List-to-Step-List

```
(ASSERT-AXIOMS
  [[MERGE-PLAN-INSTANCES ?PLAN-TYPE*T-LISP
                         ?PLAN-STEPS*T-LIST
                         ((?PLAN-TYPE . ?OLD-PLAN-STEPS*T-LIST)
                           . ?MORE-OLD-PLANS*T-LIST)
                         ?MERGED-PLANS*T-LIST]
   <RPRS [COND ([MERGE-STEPS ?PLAN-STEPS
                             ?OLD-PLAN-STEPS
                             ?MERGED-STEPS*T-LIST]
                [UNIFY ((?PLAN-TYPE . ?MERGED-STEPS) . ?MORE-NEW-PLANS)
                       ?MERGED-PLANS]
                [MERGE-PLAN-INSTANCES ?PLAN-TYPE
                                      ?PLAN-STEPS
                                      ?MORE-OLD-PLANS
                                      ?MORE-NEW-PLANS])
               ([WIN]
                [MERGE-PLAN-INSTANCES ?PLAN-TYPE
                                      ?PLAN-STEPS
                                      ?MORE-OLD-PLANS
                                      ?MERGED-PLANS])]]

  [[MERGE-PLAN-INSTANCES ?PLAN-TYPE*T-LISP ?PLAN-STEPS*T-LIST NIL NIL]
   <RPRS])
```

Figure 6.12: Rhet Support Code for Parsing Plan Proofs - Merge-Plan-Instances

```
(ASSERT-AXIOMS
  [[MERGE-STEPS ((?STEP*T-ATOM ?OBS*T-ANYTHING) . ?MORE-STEPS)
                ?OLD-PLAN-STEPS*T-LIST
                ?MERGED-STEPS*T-LIST]
   <RPRS [MERGE-STEP ?STEP ?OBS ?OLD-PLAN-STEPS ?UPDATED-STEPS*T-LIST]
         [MERGE-STEPS ?MORE-STEPS ?UPDATED-STEPS ?MERGED-STEPS]]
  [[MERGE-STEPS NIL ?OLD-STEPS*T-LIST ?OLD-STEPS] <RPRS])
```

Figure 6.13: Rhet Support Code for Parsing Plan Proofs - Merge-Steps

```
(ASSERT-AXIOMS
  [[MERGE-STEP ?STEP*T-ATOM
               ?OBS*T-ANYTHING
               ((?OLD-STEP*T-ATOM ?OLD-OBS*T-ANYTHING)
                  . ?MORE-OLD-STEPS*T-LIST)
               ((?OLD-STEP ?OLD-OBS) (?STEP ?OBS) . ?MORE-OLD-STEPS)]
   <RPRS ]      ; may eventually want to check compatibility if
                ; ?old-step is eq to ?step
  [[MERGE-STEP ?STEP*T-ATOM ?OBS*T-ANYTHING NIL (?STEP ?OBS)] <RPRS])
```

Figure 6.14: Rhet Support Code for Parsing Plan Proofs - Merge-Step

```
(DEFRHETPRED HAS-STEP-RECURSIVE (&BOUND ?STEP-TYPE*T-LISP
                                        ?STEP-CF*T-ANYTHING
                                &ANY ?PLAN-TYPE*T-ATOM
                                &UNBOUND ?ASSERTION-LIST*T-LIST)
  "True for a ?step-type in rplan ?plan-type making assertions in
?assertion-list, whose CAR is a plan type, and CADR is the step number in
the plan to be asserted equal to the step,  and whose CDDR is another
assertion-list (whose object is this just defined thing.)

i.e. (t-foo-plan s1 t-bar-plan s2) means that the passed step is s1 in
t-foo-plan, and this t-foo-plan instance is s2 of a t-bar-plan."
   )  ;nothing declared here, since our RHSs need different LHSs...

(ASSERT-AXIOMS
  [[HAS-STEP-RECURSIVE ?STEP-TYPE*T-LISP
                       ?STEP-CF*T-ANYTHING
                       ?RPLAN-TYPE*T-ATOM
                       (?RPLAN-TYPE*T-ATOM
                          (?STEP*T-ATOM ?STEP-CF*T-ANYTHING))]
   <RPRS [HAS-STEP :RPLAN ?RPLAN-TYPE ?MATCHED-STEP-TYPE*T-LISP ?STEP]
         [NOT [TYPE-RELATION ?STEP-TYPE :DISJOINT ?MATCHED-STEP-TYPE]]]

  [[HAS-STEP-RECURSIVE ?STEP-TYPE*T-LISP
                       ?STEP-CF*T-ANYTHING
                       ?RPLAN-TYPE*T-ATOM
                       ?A-LIST*T-LIST]
   <RPRS [HAS-STEP :CPLAN
                   ?CPLAN-TYPE*T-LISP
                   ?MATCHED-STEP-TYPE*T-LISP
                   ?STEP*T-ATOM]
         [NOT [TYPE-RELATION ?STEP-TYPE :DISJOINT ?MATCHED-STEP-TYPE]]
         [HAS-STEP-RECURSIVE ?CPLAN-TYPE
                             (?CPLAN-TYPE
                                (?STEP*T-ATOM ?STEP-CF*T-ANYTHING))
                             ?RPLAN-TYPE
                             ?A-LIST]])
```

Figure 6.15: Rhet Code for Examining KB - Has-Step-Recursive

```
(DOLIST (CURRENT-PROOF-ENTRY (UNLESS (HNAME:RVARIABLE-P CANDIDATES)
                                     CANDIDATES))
  ;; generate a new context to make the attempt in.
  (LET ((TEST-CONTEXT (PROG1 (STRING (GENSYM "RPRS-Test-Context")))))
    (CREATE-UCONTEXT TEST-CONTEXT UCONTEXT)
    (LET* ((*DEFAULT-CONTEXT* (UCONTEXT TEST-CONTEXT))
           ;; so adds and proof done in this new context.
           (RHET-TERMS:*CURRENT-CONTEXT* *DEFAULT-CONTEXT*)
           RPLAN-INSTANCE
           (ABORT T))
      (DECLARE (SPECIAL *DEFAULT-CONTEXT* RHET-TERMS:*CURRENT-CONTEXT*))
      ;; for this CURRENT-PLAN instantiate it.
      (CATCH 'RPRS-CONTRADICTION-FOUND
        ;; add equalities for the EVENTs and if we actually finish,
        ;; we have succeeded in showing by construction
        ;; a consistent plan which "explains" all the observed events.
        (LABELS ((CREATE-PLAN (PLAN-TYPE STEP-ENTRIES)
                   (LET ((INSTANCE (DEFINE-INSTANCE
                                      (CONS-RHET-FORM
                                         (GENSYM "CUR-PLAN"))
                                         PLAN-TYPE)))
                     (DOLIST (STEP-ENTRY STEP-ENTRIES)
                       (ADD-EQ (IF (CONSP (SECOND STEP-ENTRY))
                                   (CREATE-PLAN
                                      (CAR (SECOND STEP-ENTRY))
                                      (CDR (SECOND STEP-ENTRY)))
                                   (SECOND STEP-ENTRY))
                               (CONS-RHET-FORM (FIRST STEP-ENTRY)
                                               INSTANCE)
                               :HANDLE-ERRORS T))
                     INSTANCE)))
          (SETQ RPLAN-INSTANCE (CREATE-PLAN
                                  (CAR CURRENT-PROOF-ENTRY)
                                  (CDR CURRENT-PROOF-ENTRY))))
        (SETQ ABORT NIL))
      (IF ABORT
          (DESTROY-UCONTEXT TEST-CONTEXT)  ; error
          (PUSH (LIST RPLAN-INSTANCE TEST-CONTEXT) RETURN-VALUE)))))
RETURN-VALUE))
```

Figure 6.16: Generating Explanations - Building Instances in Rhet

Chapter 7

RPRS Function Reference

Define-Action-Type *Type* &Rest *Args* &Key *(Parent 'T-Action)*

Defines an RPRS compatible Action type, which may appear as a constructor function in a step. Warning: be sure to define these *before* use, otherwise types won't get updated properly (and RPRS will not find any applicable plans with steps). Args are as one would pass to Rhet's Define-Functional-Subtype. Actions may NOT use the STEPS relation, however.

Define-Cplan-Type *Type* &Rest *Args* &Key *(Parent 'T-Cplan) Steps*

Defines an RPRS compatible plan type, which may *not* be returned by the recognizer as a "complete" or "Recognized" plan, rather it is a "Constituent" plan. It is intended to appear as a step in some other constituent or recognized plan, but may not be an observed action (*i.e.* an argument to **Explain-Observations**). The format of the Steps argument is special: it is a list of entries indicating the constructor functions for the actions (or other Cplans) that are the steps to implement this Cplan. An entry is a cons, whose car is a keyword that is the accessor for the step, and whose cdr is this constructor function. For example,

(7.1)
```
'#[ ((:S-1 . [C-make-noodles [F-AGENT ?SELF]
                             ?TIME-S-1*T-TIME
                             ?RESULT*T-NOODLE-DISH])
     (:S-2 . [C-make-sauce [F-AGENT ?SELF]
                           ?TIME-S-2*T-TIME])) #]
```

might be the Steps entry for some Cplan. Note the use of Rhet's #[operator; this is to assure that all references to ?self are identical in the list. To reference a particular step, say, in the initializations or constraints field, one uses the accessor (without the colon) as the head of a form, *e.g.* [S-1 ?self] would refer to the step with accessor :S-1 of my type. By default, STEPS are defined as a MERGE relation in Rhet, which

means that subtypes may define new Steps, and if the accessors are the same, the new definition replaces[1] the old, while other steps are simply inherited.

Define-Rplan-Type *Type* &Rest *Args* &Key *(Parent 'T-Rplan) Steps*

Defines an RPRS compatible plan type, which may be returned by the recognizer as a "complete" or "Recognized" plan. It may not be used as a step in either a constituent or recognized plan, nor may it be an observed action (*i.e.* an argument to Explain-Observations)[2]. For an explanation of the Steps argument, see Define-Cplan-Type, above.

Explain-Observations &Rest *Actions* &Key *Ucontext*

This returns a list of lists, whose cars are possible Plan instances that describe all of the passed Actions, and whose cdr is the name of the Rhet user context that this instance is instantiated in. Each action object must be an instance of a subtype of T-Action. The Ucontext, if present, should be a Rhet user context defined directly to Rhet, or one passed back from a previous call to Explain-Observations.

Note that one can make subsequent calls to Explain-Observations passing in one of these returned contexts, this constrains the new explanations to be consistent with the previously discovered plan (providing, of course, that such links are properly added, e.g. that some step in this new plan is equal to the previously discovered plan).

Show-Explanation *(Plan-Instance Context)*

Given a cons, such as an element of the result of Explain-Observations, this pretty prints out the explanation for the user, that is, it shows the Rhet type of the recognized plan, it's steps, *etc.*.

Reset-RPRS

This function resets the RPRS system (and the underlying Rhet and TEMPOS systems) to their just-loaded and initialized condition.

---

[1] Actually specializes, since the type of the new constructor must be a subtype of the old.

[2] Future implementation of RPRS could lift the restriction on allowing Rplans to be steps in other Rplans, but the code is much more efficient and easy to understand with this restriction.

Appendix A

Installing and Running RPRS

RPRS is supplied on the same tape as the Rhet system and TEMPOS. After restoring the distribution, just do ":Load System RPRS" and then RPRS functions will be exported to the RHET-USER package. It can then be easily used with the Rhet window interface. Be sure to use ":Reset RPRS"[1] instead of ":Reset Rhetorical"[2], however, or else not only will TEMPOS not be reset, but Rhet's will not be correctly configured to work with the RPRS system and will cause strange results[3].

[1] Reset-RPRS

[2] Reset-Rhetorical

[3] Resetting RPRS causes a number of Rhet's options to be changed from the default setting that is restored after resetting Rhet.

Bibliography

[Allen, 1983] James F. Allen, "Maintaining Knowledge About Temporal Intervals," *Communications of the ACM*, 26(11):832–843, 1983.

[Allen, 1990] James F. Allen, "Unknown at press time," Technical report, University of Rochester, Computer Science Department, 1990, Forthcoming; presented at 1988 workshop on Principles of Hybrid Reasoning, 21 August, 1988 St. Paul, Minnesota.

[Allen and Koomen, 1983] James F. Allen and Johannes A. Koomen, "Planning Using a Temporal World Model," In *Proccedings 8th IJCAI*, pages 741–747, Karlsruhe, W. Germany, August 1983.

[Allen and Miller, 1989] James F. Allen and Bradford W. Miller, "The Rhetorical Knowledge Representation System: A User's Guide," Technical Report 238 (rerevised), University of Rochester, Computer Science Department, March 1989.

[Kautz, 1987] Henry A. Kautz, "A Formal Theory of Plan Recognition," Technical Report 215, University of Rochester, Computer Science Department, May 1987, PhD Thesis.

[Koomen, 1988] Johannes A.G.M. Koomen, "The TIMELOGIC Temporal Reasoning System," Technical Report 231 (revised), University of Rochester, Computer Science Department, October 1988.

[Koomen, 1989] Johannes A.G.M. Koomen, "Reasoning About Recurrence," Technical Report 307, University of Rochester, Computer Science Department, July 1989, PhD Thesis.

[Miller, 1989] Bradford W. Miller, "Rhet Programmer's Guide," Technical Report 239 (rerevised), University of Rochester, Computer Science Department, March 1989.

[Nilsson, 1980] N. J. Nilsson, *Principles of Artificial Intelligence*, Morgan Kaufmann, 1980.

Index